Components In Layered Queueing Networks

D. McMullan

1.0 Introduction

Layered queuing networks (LQNs) have been used to model and evaluate performance of many different systems such as web servers[1] and network file systems[2]. Many of these common systems may occur as parts of larger systems. If the same subsystem occurs many times within a larger system, it must be modeled repeatedly. As a result, building an LQN model of a large system that has a number of identical or similar subsystems becomes tedious and error-prone, especially if the subsystems are also large or have complex interactions with their environments. Evaluating alternative LQN models where particular subsystems are replaced by other subsystems requires the larger model to be reconstructed for each alternative. This is also tedious. To address these concerns, the concept of an LQN component is introduced.

A component is a pre-constructed LQN sub-model that can be plugged into another LQN model. A component is different from a normal model in that it declares a public interface. It is through its interface that a component is bound to its external environment. In order to plug a component into a larger model, knowledge of the internal structure of the component's model is not needed. All that is needed is knowledge of the component's interface. The interface consists of the services the component provides, the requests it makes to its external environment and the processors that may be replaced by external processors.

Sub-systems of the same type are not necessarily identical. For example, consider two web servers at different sites both running the same server program on the same type of equipment.

One site may be configured to run up to 1000 http daemons at a time while the other may only run up to 50. A model of the larger system containing the web servers would include two web server components that differ only by a configuration parameter; the maximum number of daemons. To allow different configurations of components while still maintaining a single component type, component classes are introduced. Component classes are similar to classes from object-oriented programming languages. Each component is an instance of a component class. When the component is instantiated, it is parameterized to the configuration it has in the external model. By using a component class, many different components may be created from the same template. In this example, two web servers can be instantiated from the same web server component class. Each web server component has the same interface as the component class but is parameterized differently.

Using components in layered queuing models has many advantages. Large and complex layered queueing networks can be constructed quickly. Model designs stay flexible because compatible components can be interchanged easily. Components may be unit-tested in isolation of a larger model and verified components may be stored in a library for future use.

A tool called Layered Queuing Network Solver (LQNS) is used to estimate the performance of layered queuing network models[3]. Text files that describe the models are used as input files to the solver. A model is described using the Stochastic Rendezvous Network (SRVN) modeling method[4]. However, this modeling method does not include components. Therefore, it is necessary to extend the grammar of the method to incorporate components and component classes. To produce model input files that LQNS can evaluate, a tool was developed that preprocesses component-containing input files to produce standard LQNS input files. Currently this tool is unnamed.

The rest of the paper is structured as follows. Section 2 outlines the protocol and format of a component class file. Section 3 outlines the protocol for inserting components into a LQNS input file. Section 4 describes some restrictions for binding components into LQN models. Section 5 outlines an additional function of the pre-processor filtering tool that will automatically replicate certain component interface services and the additional change in SRVN grammar that is required. Finally section 6 illustrates examples of using a Linux Network File System LQN component class as a plug-in of larger LQNs.

2.0 Component Class File Format

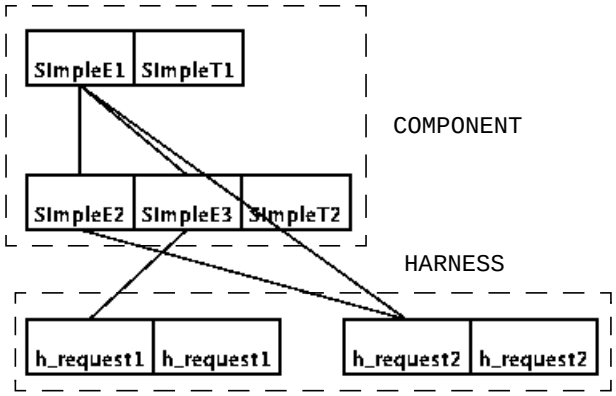
2.1 General Description

A component class file contains the description of the component class. The file name must be the same as the component class name. The file and the directory containing it must be readable by the filtering program. A component class file may include any number of comment lines which are ignored by the filtering program. Comments begin with '#'. An example component class file (SimpleMod) along with a graphical representation of the component's structure can be found in Figure 1.

```
M SimpleMod ($T1_multi = 1, $T2_multi = 1)
# Interface/Export List.
I
p Pr1 -1               #processors that can be replaced by outside processors
e SimpleE1 -1          #entries that can be bound to outside entries (sources)
r h_request1 h_request2 -1#requests that can be bound to outside entries (sinks)
-1

# General Information. This section is used when component is a stand-alone model
G
"Example of Simple Component"
0.00001
100
1
0.9
-1

# Definition of component
P 0
p Pr1 f
p Pr2 f
-1
```




```
T 0
t SimpleT1 n SimpleE1 -1 Pr1 m $T1_multi
t SimpleT2 n SimpleE2 SimpleE3 -1 Pr2 m $T2_multi
-1

E 0
s SimpleE1 $VAR1 1 0 -1
s SimpleE2 1 0 0 -1
s SimpleE3 1 0 0 -1
y SimpleE1 SimpleE3 2 0 0 -1
y SimpleE1 SimpleE2 1 1 0 -1
y SimpleE3 h_request1 1 0 0 -1
y SimpleE2 h_request2 1 0 0 -1
y SimpleE1 h_request2 1 0 0 -1
-1

# Environment Harness. This section is used for defaults when interface connections
#                      are not specified
H
T 0
t h_request1 i h_request1 -1
t h_request2 i h_request2 -1
-1
E 0
s h_request1 1 0 1 -1
s h_request2 1 0 0 -1
a SimpleE1 0.05
-1
-1
```

Figure 1: Example of Component Class File (SimpleMod)

2.2 Component Class File Sections

A component class file contains at least five major sections: Component Declaration, Interface, General Information, Component Definition, and Environment Harness. The file must include the first four sections but the Environment Harness section is optional under certain conditions. A component may contain zero or more Binding Sections.

2.2.1 Component Declaration Section

The Component Declaration section consists of a single line:

‘M’ <ComponentClass> ‘(‘<VariableList>’)’

The letter M signifies that the file is a component class file. ComponentClass is the class name of the component. It must be the same as the name of the component class file. Following the class name is a list of comma-separated variable definitions enclosed by parentheses. Variables are symbols in the Component Definition section of the component class file that may have different numeric values. All variable names begin with ‘$’. Default values for the variables are defined in the VariableList using an assignment statement.

For example: $T1_multi = 1

When a component is instantiated, the filtering program replaces variables declared in the VariableList either with parameter values assigned to the variables through the binding statement or if not bound, with the defaults values from VariableList. For a variable found in the component but not declared in VariableList, no value is substituted and the variable becomes a model variable.

Here is an example of a component declaration:

M SimpleMod($T1_multi=1, $T2_multi=1)

2.2.2 Interface Section

The Interface section specifies the parts of the component that are visible to the external component or model. Interface entities include component processors which may be replaced by external processors, entries that provide component services and entries that request services from external entries. It is through these interface entities that the component is bound to its external environment.

The Interface section has the format:

I
<ProcessorInterfaceList>
<ServiceInterfaceList>
<RequestInterfaceList>
-1

The ProcessorInterfaceList is a list of one or more component processors that may be replaced by external processors. This accommodates placing certain component tasks on processors that are external to the component when the component is instantiate and bound. ProcessorInterfaceList begins with the letter ‘p’.

The ServiceInterfaceList is a list of one or more component entries that provide services to external entries. These are the component’s services and act as the source entries of the component. ServiceInterfaceList begins with the letter ‘e’.

The RequestInterfaceList is a list of one or more entries that act as sinks to requests made by component entries. It is actually a list of the entries from the component’s environment harness

that provide services to the component. When the component is instantiated and bound, references to these entries may be replaced by external references. RequestInterfaceList begins with the letter ‘r’.

All lists end with ‘-1’. The Interface section must contain at least one of the lists and the lists may be in any order.

Here is an example of an Interface section:

```
I
p Pr1 -1                    #processors that can be replaced by outside processors
e SimpleE1 -1               #entries that can be bound to outside entries (sources)
r h_request1 h_request2 -1  #requests that can be bound to outside entries (sinks)
-1
```

2.2.3 General Information Section

The format of the General Information section of a component class file is the same as the G section of an LQN file. It is included so the component can be instantiated as a stand-alone model. This section is ignored if the component instance is being inserted in another component or LQN model.

2.2.4 Component Definition Section

The Component Definition section contains all the information necessary to define the processors, tasks, entries and activities of the component. The section consists of up to four subsections: Processor Information, Task Information, Entry Information, and Activity Information. The Activity Information section is optional. The format of these sections is similar to the format of the same sections of an LQN file, except that variables may replace numeric values. Any variables not declared in the Component Declaration section will have the component’s name inserted after the “$” character and the variable becomes a variable in the final LQN file. For example, if the

Component Definition section of a component named “T3” has a variable “$VAR1” that is not assigned a value in the Component Declaration section, the variable becomes “$T3_VAR1” in the model.

Tasks that contain entries declared as interface sources in the Interface section may not be defined as reference tasks. Definitions of processors, tasks, entries and activities that are not part of the component proper but are part of its environment harness are not included in the Component Definition section. These entities are used to drive the component and are declared in the Environment Harness section.

2.2.5 Environment Harness Section

The Environment Harness of a component is used to ‘drive’ the component when the component is instantiated as a stand alone model. A environment harness for a component mainly consists of open arrival rates to the source interface entries and of tasks containing the sink interface entries of the component. These tasks may have their own processors or may be pure delay server tasks.

It is possible for a component to have no environment harness. This would occur when the source interface entries are driven by other component entries or activities and when there are no requests for external service. In this case, the Environment Harness section is not included in the component class file.

The Environment Harness section is also used when the interface bindings of the instance of the component are not fully specified when it is created. In this case, open arrival rates of the source entries and sink tasks may be obtained from the environment harness section by the filtering program and inserted into the final model.

The Environment Harness section consists of up to four subsections: Processor Information, Task Information, Entry Information, and Activity Information. The Activity Information section is optional. Because it is possible to have a environment harness with no processors, the Processor Information section is also optional. If the component is instantiated as a stand-alone model, the filtering program will extract the information from the Environment Harness section and combine it with the Component Definition section to produce a executable LQN model.

The Environment Harness section begins with an ‘H’ and ends with ‘-1’. By convention, the names of all tasks and entries in the Environment Harness section begin with the letter ‘h’. Here is an example of a environment harness with no processors or activities:

```
H
T 0
t h_request1 i h_request1 -1
t h_request2 i h_request2 -1
-1
E 0
s h_request1 1 0 1 -1
s h_request2 1 0 1 -1
a SimpleE1 0.05
-1
-1
```

2.2.6 Binding Section

Tasks within components may also be replaced by instances of other components. This allows components to be nested. Caution must be exercised when nesting components to make sure that circular or recursive references are avoided. Full descriptions of how Binding sections are used and of the format are given in Sections 3.0 and 3.1.

3.0 LQN File Format

One component is used to replace one task in an LQN with a set of tasks. Therefore before a component is inserted, the LQN model must be constructed so that it includes the original task. To replace the task with a component, a Binding section for that task is appended to the end of the LQN text file description. This Binding section declares and defines the component type and its bindings to the LQN model. There must be a separate Binding section for every task replaced by an instance of a component class.

3.1 Binding Section

The Binding section declares and defines the replacement of a single task by a component. It is used to create a new instance of a component class and to define processor replacements, service bindings, and request bindings that are declared in the Interface section of the component. The section begins with the letter ‘B’ and ends with ‘-1’.

The first line has the format:

‘B’ <Taskname> <ComponentClass> ‘(‘ <ParameterList> ‘)’

Taskname is the name of the task in the model that is replaced by the component. This name will become the instantiated component’s name. To ensure unique names in the expanded LQN model, all names within the instantiated component will be prefixed by the Taskname.

ComponentClass is the name of the component class that replaces the task. It is also the file name of the file containing the component definition.

ParameterList is a list of comma-separated variable definitions to be applied to the instance of the component. The variables in the list must be variables of the component class and the assigned

values may be numbers or variables. The order in which the variables are defined is not important. Any component variables not defined in this list will be assigned their default values in the instance of the component.

Here is an example of the creation of an instance of SimpleMod to replace task T3. The component part names begin with ‘T3_’ , the component variable ‘$T1_multi’ is set to 2 and all other variables are assigned their default values:

B T3 SimpleMod($T1_multi = 2)

Following the first line of the Binding section is one or more specific bindings. The binding types are the same as the Interface types of a component as described in the Interface section of the Component Class File Format (Section 2.2.2). The general format of each line is:

<BindingType> <ModelPartName> <ComponentPartName>

BindingType can be ‘e’ for service, ‘r’ for request or ‘p’ for processor. ModelPartName is the name the part has in the model. ComponentPartName is the name the component’s interface part has in the instantiated component, that is, the part name is prefixed by the component’s instance name.

A service binding binds one of the replaced task’s entries to one of the component’s interface service entries. It has the form:

‘e’ <TaskService> <ComponentServiceInterface>

TaskService is the name of the service entry in the task being replaced by the component. ComponentServiceInterface is the name of the entry in the component that will satisfy the

requests to the replaced TaskService. In other words, <TaskService> ‘is provided by’ <ComponentServiceInterface>.

A request binding binds one of the model’s entries to one of the component’s interface request entries. It has the form:

‘r’ <ModelService> <ComponentRequestInterface>

ModelService is the name of the entry in the model that provides service for requests made to the ComponentRequestInterface. That is, <ModelService> ‘services requests made to’ <ComponentRequestInterface>.

A processor binding binds one of the model’s processors to one of the component’s processors. It has the effect of moving part of the component to the external model. A processor binding has the form:

‘p’ <ModelProcessor> <ComponentProcessorInterface>

ModelProcessor is the name of the processor in the model that replaces the component’s processor, ComponentProcessorInterface. That is, <ModelProcessor> ‘replaces’ <ComponentProcessorInterface>.

Here is an example of a Binding section;

```
B T3 SimpleMod($T1_multi = 2)
e e3 T3_SimpleE1
r e4 T3_h_request1
r e5 T3_h_request2
p P2 T3_Pr1
-1
```

The example shows that model task T3 is replaced by a SimpleMod component. The value of the $T1_multi variable of the SimpleMod instance is set to 2. Other variables are set to their default values. Service provided by model entry e3 is provided by component entry T3_SimpleE1. Model entries e4 and e5 service requests made to the component's interface request entries T3_h_request1 and T3_h_request2, respectively. Finally, model processor P2 replaces component processor T3_Pr1.

Example LQN files along with a graphical representation of the model structure of an LQN file with no component and of an LQN file where T3 is substituted by an instance of a component (SimpleMod) can be found in Figures 2 and 3, respectively.

The final LQN file that is produced by the filtering program can be found in Figure 4.

```
G
"Example with no component"
0.00001
100
1
0.9
-1

P 3
p P1 f
p P2 f
p P3 f
-1

T 5
t T1 r e1 -1 P1
t T2 r e2 -1 P2
t T3 n e3 -1 P3
t T4 n e4 -1 P3
t T5 n e5 -1 P3
-1

E 5
s e1 1 0 0 -1
y e1 e3 1 0 0 -1
s e2 1 1 0 -1
y e2 e3 1 0 0 -1
s e3 1 0 0 -1
y e3 e4 1 0 0 -1
y e3 e5 1 0 0 -1
s e4 1 0 0 -1
s e5 1 0 0 -1
-1
```

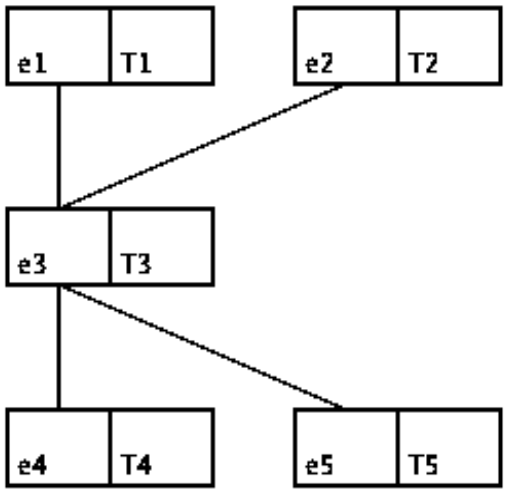


Figure 2: LQN file with no component

```
G
"Example with T3 replaced by an instance of SimpleMod"
0.00001
100
1
0.9
-1

P 3
p P1 f
p P2 f
p P3 f
-1

T 5
t T1 r e1  -1 P1
t T2 r e2  -1 P2
t T3 f e3  -1 P3
t T4 f e4  -1 P3
t T5 f e5  -1 P3
-1

E 5
s e1 1 0 0 -1
y e1 e3 1 0 0 -1
s e2 1 1 0 -1
y e2 e3 1 0 0 -1
s e3 1 0 0 -1
y e3 e4 1 0 0 -1
y e3 e5 1 0 0 -1
s e4 1 0 0 -1
s e5 1 0 0 -1
-1

B T3 SimpleMod($T1_multi=2)
e e3 T3_SimpleE1
r e4 T3_h_request1
r e5 T3_h_request2
p P2 T3_Pr1
-1
```

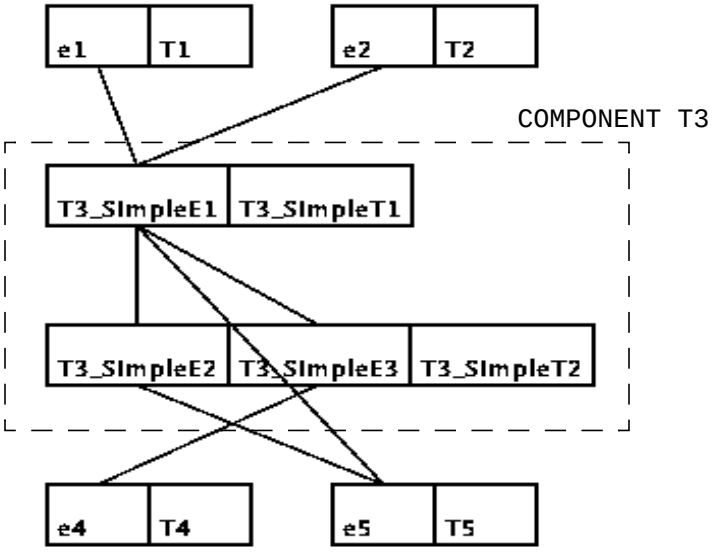


Figure 3: LQN file where T3 is substituted by an instance of a component (SimpleMod)

```
 G
“Example with T3 replaced by an instance of SimpleMod”
1.0E-5
100
1
0.9
-1

P 4
p P1 f
p P2 f
p P3 f
p T3_Pr2 f
-1

T 6
t T1 r e1  -1 P1
t T2 r e2  -1 P2
t T3_SimpleT1 f T3_SimpleE1  -1 P2 m 2
t T3_SimpleT2 f T3_SimpleE2 T3_SimpleE3  -1 T3_Pr2
t T4 f e4  -1 P3
t T5 f e5  -1 P3
-1

E 7
s e1 1.0 0.0 0.0 -1
y e1 T3_SimpleE1 1.0 0.0 0.0 -1
s e2 1.0 1.0 0.0 -1
y e2 T3_SimpleE1 1.0 0.0 0.0 -1
s T3_SimpleE1 $T3_VAR1 1.0 0.0 -1
y T3_SimpleE1 T3_SimpleE2 1.0 1.0 0.0 -1
y T3_SimpleE1 T3_SimpleE3 2.0 0.0 0.0 -1
y T3_SimpleE1 e5 1.0 0.0 0.0 -1
s T3_SimpleE2 1.0 0.0 0.0 -1
y T3_SimpleE2 e5 1.0 0.0 0.0 -1
s T3_SimpleE3 1.0 0.0 0.0 -1
y T3_SimpleE3 e4 1.0 0.0 0.0 -1
s e4 1.0 0.0 0.0 -1
s e5 1.0 0.0 0.0 -1
-1
```

Figure 4: Final LQN file produced by filtering program

4.0 Binding Compatibility

It is important to ensure that the bindings of a component are compatible with the environment of the task it is replacing. Therefore, some requirements must be met before an instance of a component can be installed to its external model.

A component must be bound to the same number of sources and sinks in the external model that there are for the original task. In other words, the number of component service bindings must equal the number of entries in the task that have callers and the number of component request bindings must equal the number of different call destinations of the entries and activities of the task.

To prevent ambiguity between call destinations that could occur when components are nested, there must be a one-to-one binding ratio between component interface items and external model items. In other words, each component interface item can only have one binding definition to an external model item and any external model item can only have one binding definition to a component item.

While a component may not have more bindings than the number of its interface items, it may have fewer. If a component interface item other than a processor does not have a binding definition, the appropriate part of the component's environment harness is used. A warning is issued by the filtering program for each unbound component service or request.

5.0 Automatic Replication of Component Interface Services

Occasionally, it is desirable to replicate parts of the interface of a component and place them on resources external to the component. Consider a task representing a file server that has a num-

ber of heterogeneous clients. If the task is replaced by a file server component where part of the component's service interface should reside on each of its clients, it is necessary to replicate the component services so that each client interacts with its own copy of the service. Otherwise, the server component can only serve one client. One way to accommodate this situation is to construct the component so that the number of component service interface entries corresponds to the number of heterogeneous clients. Processor interface bindings can be defined so that each service interface task's processor is replaced by a client's processor. This will move the component task with the service interface entry and any other task that uses the same interface processor to the client's processor.

While the above method is useful, it is not very flexible because the number of clients must be known at the time the component class file is constructed. A more flexible method would provide a way of indicating that the component's service interface should be replicated at the time the component is instantiated and bound and that each external client's processor should be automatically assigned to the tasks of each replica. As a result, a unique copy of the component's interface would be located on each client.

Because other non-interface component tasks may also have to be relocated on each external client's processor, those tasks should also be assigned to the same interface processor as the service interface tasks. By replicating the interface processor, both sets of tasks are replicated as well. Because the component processor replicas are automatically replaced by the client processors, a unique copy of each set of component tasks are relocated to each client's processor.

To indicate that the service interface of an instantiated component should be replicated to accommodate as many clients as necessary and that the replicas be moved to the clients' processors, an asterisk(*) should be inserted after the 'B' marker of the Binding Section. The line would have the format:

'B*' <Taskname> <ComponentClass> '(' <ParameterList> ') '

The number of replicas created is equal to the number of unique processors of the clients that use services of the original task (Taskname). To keep the names of the tasks, entries and activities of each replica unique, a one-based index number is appended to every name in the replica. The clients of the original task are redirected to their copy of the component interface services based on the alphabetical order of their processor's name and all the tasks in the replica are placed on the client's processor.

Using the previous LQN file example where T3 is replaced by an instance of SimpleMod but now with service interface replication, the Binding Section would appear as follows:

```
B* T3 SimpleMod($T1_multi = 2)
e e3 T3_SimpleE1
r e4 T3_h_request1
r e5 T3_h_request2
-1
```

In the LQN model, task T3 has two different clients, e1 and e2, located on separate processors, P1 and P2. The interface service of SimpleMod, SimpleE1, and its task, SimpleT1, are replicated once. Calls from e1 to e3 are redirected to T3_SimpleE1_1 and calls from e2 to e3 are redirected to T3_SimpleE1_2. Task T3_SimpleT1_1 is on processor P1 and task T3_SimpleT1_2 is on processor P2. The LQN file with the binding section and the final filtered LQN file can be found in Figures 5 and 6, respectively.

```
G
"Example with T3 replaced by an instance of SimpleMod with replication"
0.00001
100
1
0.9
-1

P 3
p P1 f
p P2 f
p P3 f
-1

T 5
t T1 r e1  -1 P1
t T2 r e2  -1 P2
t T3 f e3  -1 P3
t T4 f e4  -1 P3
t T5 f e5  -1 P3
-1

E 5
s e1 1 0 0 -1
y e1 e3 1 0 0 -1
s e2 1 1 0 -1
y e2 e3 1 0 0 -1
s e3 1 0 0 -1
y e3 e4 1 0 0 -1
y e3 e5 1 0 0 -1
s e4 1 0 0 -1
s e5 1 0 0 -1
-1

B* T3 SimpleMod($T1_multi=2)
e e3 T3_SimpleE1
r e4 T3_h_request1
r e5 T3_h_request2
-1
```

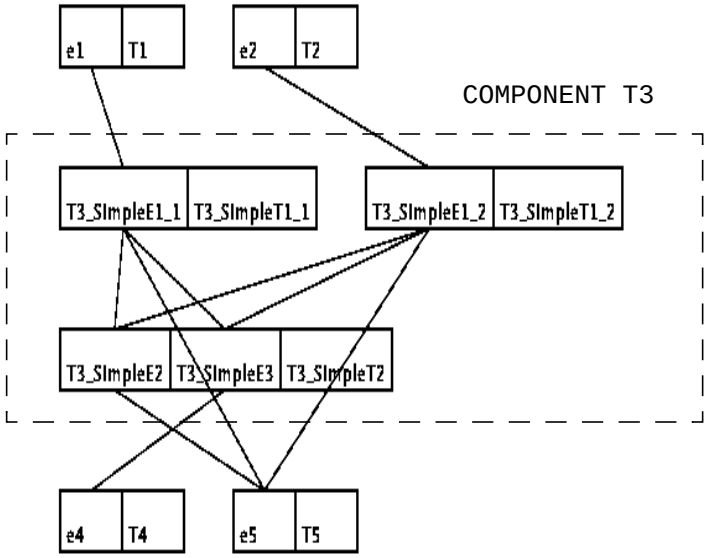


Figure 5: LQN file where T3 is substituted by an instance of a component (SimpleMod) with service interface replication

```
G
“Example with T3 replaced by an instance of SimpleMod with replication”
1.0E-5
100
1
0.9
-1

P 4
p P1 f
p P2 f
p P3 f
p T3_Pr2 f
-1

T 7
t T1 r e1  -1 P1
t T2 r e2  -1 P2
t T3_SimpleT1_1 f T3_SimpleE1_1  -1 P1   m 2
t T3_SimpleT1_2 f T3_SimpleE1_2  -1 P2   m 2
t T3_SimpleT2 f T3_SimpleE2 T3_SimpleE3  -1 T3_Pr2
t T4 f e4  -1 P3
t T5 f e5  -1 P3
-1

E 8
s e1 1.0 0.0 0.0 -1
y e1 T3_SimpleE1_1 1.0 0.0 0.0 -1
s e2 1.0 1.0 0.0 -1
y e2 T3_SimpleE1_2 1.0 0.0 0.0 -1
s T3_SimpleE1_1 $T3_VAR1 1.0 0.0 -1
y T3_SimpleE1_1 T3_SimpleE2 1.0 1.0 0.0 -1
y T3_SimpleE1_1 T3_SimpleE3 2.0 0.0 0.0 -1
y T3_SimpleE1_1 e5 1.0 0.0 0.0 -1
s T3_SimpleE1_2 $T3_VAR1 1.0 1.0 -1
y T3_SimpleE1_2 T3_SimpleE2 1.0 0.0 0.0 -1
y T3_SimpleE1_2 T3_SimpleE3 2.0 0.0 0.0 -1
y T3_SimpleE1_2 e5 1.0 0.0 0.0 -1
s T3_SimpleE2 1.0 0.0 0.0 -1
y T3_SimpleE2 e5 1.0 0.0 0.0 -1
s T3_SimpleE3 1.0 0.0 0.0 -1
y T3_SimpleE3 e4 1.0 0.0 0.0 -1
s e4 1.0 0.0 0.0 -1
s e5 1.0 0.0 0.0 -1
-1
```

Figure 6: Final LQN file with service interface replication produced by filtering program

6.0 Example of a Network File System Component

A layered queuing network model of the Linux Network File System (NFS) service was developed[2]. The model has four parts: the client, the server, the disk on the server, and the network. To convert this model into an NFS component the reference task, nfsstone, was deleted and the component interface was declared. The resulting component class file (LinuxNFS) is shown in Figure 8.

The processor interface consists of the client processor. The service interface consists of read, write and other. In the original NFS model, the reference task made read and write requests to a task on the client processor and all other NFS requests by the reference task were made by calls to the network and server directly. To encapsulate these requests within the component, a service entry called ‘other’ was placed in the ‘clientcache’ task. This entry makes the calls that the original reference task made but has no CPU demand. The request interface consists of the network service.

The environment harness consists of open arrival rates for the service interface and a network representing a 100 Mb Ethernet.

The values for the multiplicity of the client buffer cache, the server buffer cache, the client biod daemon, and the server nsfd daemon were made variables in the component class file. The default values of all variables were set to one.

```
M LinuxNFS($CLIENT_CACHE_MULTI =1,
           $NFS_IOD_MULTI = 1,
           $RPC_NFSD_MULTI = 1,
           $SERV_CACHE_MULTI = 1)
I
  p client -1
  e read write other -1
  r ether -1
-1
G “Linux NFS Component, 8K Writes, 4K NW reads” 1e-04 300 5 0.200000 -1
  P 0
  p client s 10000
  p server s 10000
  p disk f
-1
T 0
  t clientcache n read write other                -1 client m $CLIENT_CACHE_MULTI
  t nfsiod      n b_read                          -1 client m $NFS_IOD_MULTI
  t rpcnfsd     n lookup n_read n_write            -1 server m $RPC_NFSD_MULTI
  t servercache n s_read s_write                   -1 server m $SERV_CACHE_MULTI
  t disk        n d_read_4k d_read d_write_4k d_write   -1 disk
-1
E 0
  s read                50.9 -1
  s write               76.1 2.0 -1
  s other               0.0 -1
  s b_read              11.8 -1
  s lookup              415.9 -1
  s n_read              410.4 -1
  s n_write             410.4 -1
  s s_read              20.8 2.0  -1
  s s_write             39.1 2.0  -1
  s d_read_4k           10560   -1
  s d_read              6910   -1
  s d_write_4k          2440   -1
  s d_write             11780   -1
#
  y read   b_read       0.11  -1
  y read   n_read       0.124 -1
  y read   ether        0.496 -1
  y write  ether        0.0    7.0 -1
  y write  n_write      0.0    1.0 -1
  y other  lookup       1.0  -1
  y other  ether        2.0  -1
  y b_read   n_read     1.0   -1
  y b_read   ether      4.0   -1
  y lookup   s_read     0.5   -1
  y n_read   s_read     1.0   -1
  y n_write  s_write    1.0   -1
  y s_read   d_read_4k  0.00133 0.0350 -1
  y s_read   d_read     0.00084 0.0222 -1
  y s_write  d_write_4k 0       0.0558 -1
  y s_write  d_write    0       0.0061 -1
-1
H
P 0
  p network f i
-1
T 0
  t network n ether -1 network
-1
E 0
  a read     0.00001
  a write    0.000001
  a other    0.0000005
  s ether    208.0 -1
-1
-1
```

Figure 8: Linux NFS Component Class File

Two examples are used to demonstrate the use of the LinuxNFS component.

The first example uses two instances of the LinuxNFS component that represent two separate NFS servers. One is used by an application and the other is used by a database server. The LQN file before filtering is shown in Figure 9 and a graphical representation of the final model is shown in Figure 10. Tasks NFS1 and NSF2 are replaced by instances of LinuxNFS. In this model, these tasks simply act as place holders for the components. The specific attributes of the tasks and entries are not important except that each task must have three entries that are called by users and must make calls to a single destination. This enables each component instance to have three service bindings and one request binding. In this example, the attributes of the Client, Application, DBserver, and Network were chosen arbitrarily.

The second example uses one instance of the LinuxNFS component with its service interface replicated to accommodate four different client tasks. The LQN file before filtering is shown in Figure 11 and a graphical representation of the final model is shown in Figure 12.

```
G "" 1.0E-4 300 5 0.2 -1

P 5
p ClientP s 10000.0
p DBServerP s 10000.0
p NFS1P s 10000.0
p NFS2P s 10000.0
p Network f i
-1

T 6
t Application f main  -1 ClientP m 60
t Client r client  -1 ClientP m 120
t DBServer f DBmain  -1 DBServerP m 60
t NFS1 f read1 write1 other1  -1 NFS1P
t NFS2 f read2 write2 other2  -1 NFS2P
t Network f ether  -1 Network
-1

E 10
s DBmain 95.7 -1
y DBmain other2 0.31 -1
y DBmain read2 0.63 -1
y DBmain write2 0.06 -1
s client 0.01 -1
y client DBmain 0.5 -1
y client ether 3.0 -1
y client main 0.5 -1
s ether 208.3 -1
s other1 1.0 -1
s other2 1.0 -1
s main 95.7 -1
y main other1 0.31 -1
y main read1 0.63 -1
y main write1 0.06 -1
s read1 4.0 -1
y read1 ether 1.0 -1
s read2 4.0 -1
y read2 ether 1.0 -1
s write1 1.0 -1
s write2 1.0 -1
-1

B NFS1 LinuxNFS($CLIENT_CACHE_MULTI = 4000,
                $NFS_IOD_MULTI = 4,
                $SERV_CACHE_MULTI = 4000)
e other1 NFS1_other
e read1 NFS1_read
e write1 NFS1_write
r ether NFS1_ether
p ClientP NFS1_client
-1

B NFS2 LinuxNFS($CLIENT_CACHE_MULTI = 4000,
                $NFS_IOD_MULTI = 4,
                $SERV_CACHE_MULTI = 4000)
e other2 NFS2_other
e read2 NFS2_read
e write2 NFS2_write
r ether NFS2_ether
p DBServerP NFS2_client
-1
```

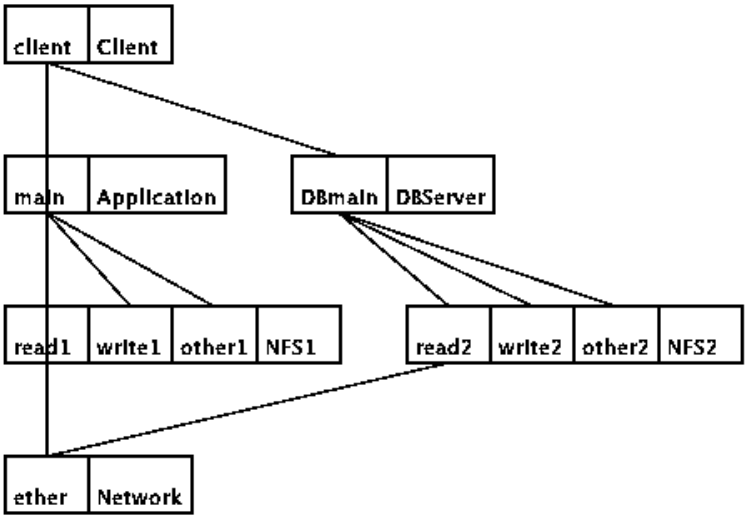


Figure 9: Example of Model with Two NFS Servers

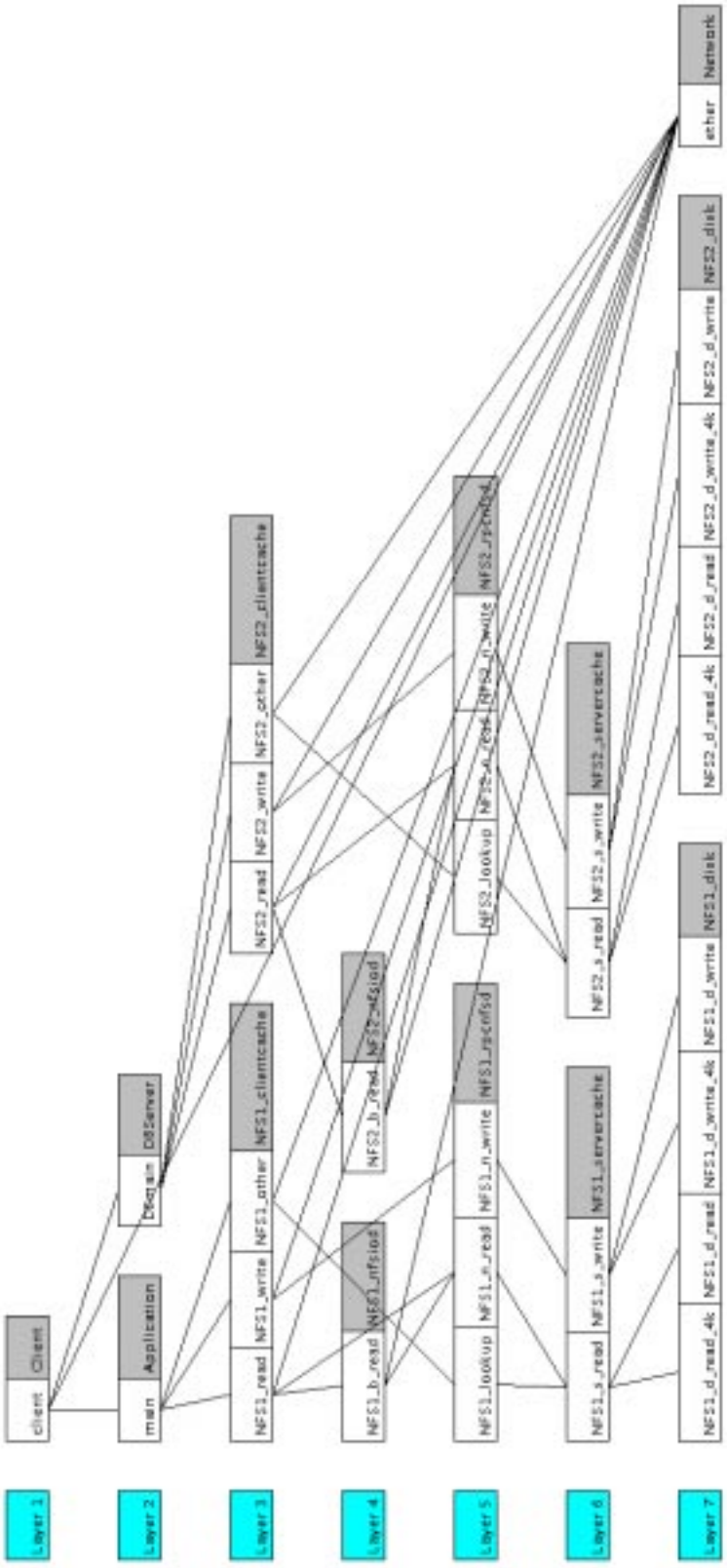


Figure 10: Expanded Version of Model with Two NFS Servers

```
G
“NFS, 8K Writes, 4K NW reads, 100MB ethernet, 4 client processors” 0.0010 200 50.0
5
-1
P 6
p client1 s 10000.0
p client2 s 10000.0
p client3 s 10000.0
p client4 s 10000.0
p server s 10000.0
p network f i
-1
T 6
t Client1 r main1  -1 client1 m 60
t Client2 r main2  -1 client2 m 60
t Client3 r main3  -1 client3 m 60
t Client4 r main4  -1 client4 m 60
t NFS f other read write  -1 server
t network f ether  -1 network
-1
E 8
s main1 95.7 -1
y main1 other 0.31 -1
y main1 read 0.63 -1
y main1 write 0.06 -1
s main2 95.7 -1
y main2 other 0.31 -1
y main2 read 0.63 -1
y main2 write 0.06 -1
s main3 95.7 -1
y main3 other 0.31 -1
y main3 read 0.63 -1
y main3 write 0.06 -1
s main4 95.7 -1
y main4 other 0.31 -1
y main4 read 0.63 -1
y main4 write 0.06 -1
s other 415.9 -1
s read 410.4 -1
y read ether 1.0 -1
s write 410.4 -1
s ether 235.2 -1
-1

B* NFS LinuxNFS($CLIENT_CACHE_MULTI = 1000,
                $NFS_IOD_MULTI = 4,
                $SERV_CACHE_MULTI = 1000)
e other NFS_other
e read NFS_read
e write NFS_write
r ether NFS_ether
-1
```

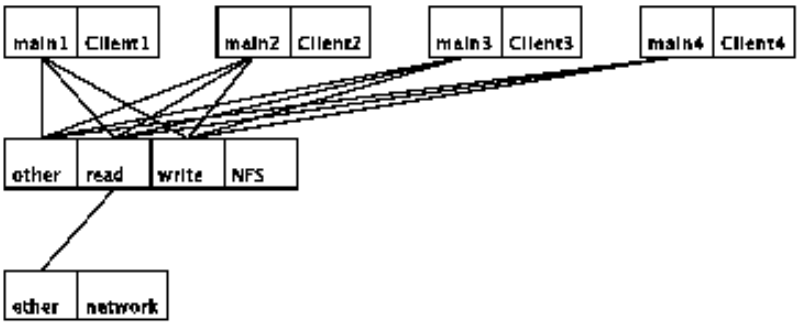


Figure 11: Example of NFS Model with replication for 4 clients

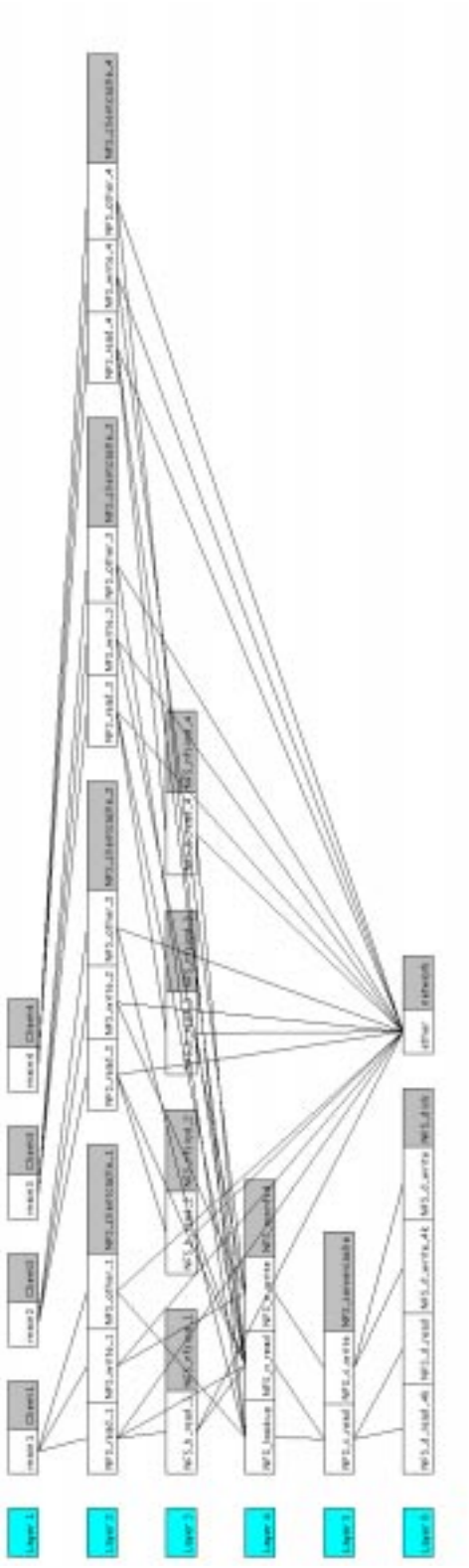


Figure 12: Expanded Version of NFS Model with replication for 4 clients

7.0 References

[1] J. Dilley, R. Friedrich, T. Jin and J. Rolia. Web server performance measurement and modeling techniques. *Performance Evaluation*, 33:5-26, 1998.

[2] Greg Franks and Murray Woodside. A Re-Usable Plug-in Performance Model of the Linux 2.0 Network File System. CarletonUniversity, December 1999.

[3] R.G. Franks, A. Hubbard, S. Majumdar, J.E. Neilson, D.C. Petriu, J. Rolia and C.M. Woodside. A Toolset for Performance Engineering and Software Design of Client-Server Systems. *Performance Evaluation*, vol. 24, no. 1-2, pp 117-135, November, 1995.

[4] D.C. Petriu, R.G Franks and A. Hubbard. SRVN Input File Format. Carleton University, November 24, 1998.